Computer, Network, and Java Security
Introduction

- Need for Computer/Internet security
  - Consumers buying products, trading stocks, and banking online
  - Credit-card, social security, and confidential business information exchanged
  - Security attacks
    - Data theft and hacker attacks
    - Wireless transmissions easier to intercept

- Security fundamentals
  - *Privacy*: no third party
  - *Integrity*: information unaltered
  - *Authentication*: proving identities
  - *Non-repudiation*: legal proof of message received
  - Availability: Network stays in operation continuously
Computer Security: General Rules

- Limits of computer security
  - Absolute computer security is not feasible
  - Given unlimited resources any form of security can be broken
  - Objective: cost for breaking a system must far outweigh rewards

- End-to-end security
  - In multitier systems, each tier must have its own security
  - Security is as strong as the weakest link

- Complex vs. Simple systems
  - Complex systems: high cost of design and implementation
  - Simple systems: Easier understood, better analyzed

- Always required
  - Security must be an integral part of a system design
Types of Threats

- Secrecy Attacks
  - Attempts to steal confidential information

- Integrity Attacks
  - Attempts to alter information with malicious intent

- Availability Attacks
  - Attempts to disrupt a system’s normal operation
Example of Attacks

- **Brute force**
  - Involves searching every key until the right one unlocks the system

- **Trojan Horse**
  - Involves planting an enemy program as an insider in such a way that it is not apparently noticeable

- **Person-in-the-middle attack**
  - Attacker intercepts the communication between two parties without their knowledge
Protections

- Network related:
  - Firewalls
  - Virtual Private Networks

- Cryptography
  - Design of algorithms for encrypting and decrypting information
    - Plaintext: unencrypted data
    - Ciphertext: encrypted data
    - Key: used by sender and receiver to encrypt and decrypt message
  - Provides confidentiality (only the intended recipient can make sense of the message)
Protections (cont’d)

- Authentication
  - Confirms user’s identity (e.g. passwords, smart cards, biometrics, etc.)

- Authorization
  - After authentication, access to the user is governed by an access control policy

- Auditing and logs
  - Keeping a record of resource access that were granted or denied can serve in preventing or analyzing a break-in
## Security Layered Architecture

<table>
<thead>
<tr>
<th>Layer</th>
<th>Security Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Encryption, Authentication, Authorization, JVM security</td>
</tr>
<tr>
<td>Transport</td>
<td>Encryption, Authentication, Authorization</td>
</tr>
<tr>
<td>Network</td>
<td>SSL/TLS</td>
</tr>
<tr>
<td>Link Layer</td>
<td>IPSEC, firewalls, VPN</td>
</tr>
<tr>
<td>PHY</td>
<td>WEP, PPP LCP</td>
</tr>
<tr>
<td></td>
<td>Securing cabling, avoiding Interference, Spread Spectrum</td>
</tr>
</tbody>
</table>
Cryptanalysis

- Even if keys are secret, it is possible to compromise the security of a system.

- Cryptanalysis: trying to decrypt ciphertext without knowledge of the decryption key
  - Cryptanalytic attacks

- Attacks can be reduced if proper key management structures are in place and keys use expiration dates.
General Security Considerations

- Know your enemy
- Identify assumptions and weaknesses
- Control secrets
- Remember human factors
- Limit the scope of access
- Understand your environment
- Remember physical security
- Make security pervasive
Java Security Extensions

- If you are using JDK 1.3.x, download
  - JCE 1.2.2
  - JAAS 1.0 class libraries
  - JSSE 1.0.3
- Copy *.jar to C:\jdk1.3.1\jre\lib\ext
- Insert the follow two lines to C:\jdk1.3.1\jre\lib\security\java.security after the line `security.provider.2=...`

```
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
```
Cryptography Algorithms

- Based on the secrecy of the algorithm (Ancient Ciphers mostly):
  - Substitution ciphers: given letter replaced by different letter. Example: Rot13, rotates a character in the message by 13 positions
  - Transposition ciphers: letter ordering shifted

- Based on the secrecy of the key (Modern Algorithms):
  - One-way hash functions
  - Symmetric ciphers
  - Asymmetric ciphers
1. One-way hash functions

- Given input message $M$ of any length, compute $h = H(M)$ to produce a hash value $h$ of length $m$

- Properties:
  - Given $M$, it is easy to compute $h$
  - Given $h$, it is hard to compute $M$ such that $H(M) = h$
  - Given $M$, it is hard to find a message $M'$, such that $H(M) = H(M')$

- Useful to produce fingerprints
  - RSA’s MD4, MD5 (RFC 1321, 1992)
    - MD=Message Digest
    - RSA=Ron Rivest, Adi Shamir, and Leonard Adlemaen
    - Produce a 128-bit hash
  - NIST and NSA’s SHA, SHA-1 (1994)
    - SHA=Secure hashing algorithm
    - Produces a 160-bit hash used in the Digital Signature Algorithm (DSA)
### Example: MD5

<table>
<thead>
<tr>
<th>Original Message</th>
<th>Hash value (in hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a quick brown fox jumped over a lazy dog</td>
<td>13b5eeb338c2318b790f2ebcccb91756f</td>
</tr>
<tr>
<td>a quick blue fox jumped over a lazy dog</td>
<td>32c63351ac1c7070ab0f7d5e017dbcea</td>
</tr>
<tr>
<td>a quick brown dog jumped over a lazy fox</td>
<td>a4c3b4cd38ade6b5e2e101d879a966f5</td>
</tr>
</tbody>
</table>
MD5/SHA in Java

```java
import java.security.*;
import java.io.*;

public class md5 {

    public static void main(String args[]) {

        if (args.length != 1) {
            System.out.println("Usage: java md5 <your text>");
            System.exit(1);
        }

        try {
            // Create an output file "digest"
            FileOutputStream digestStream = new FileOutputStream("digest");
            // Use the MD5 algorithm. SHA will work as well
            MessageDigest md=MessageDigest.getInstance("MD5");
            byte buf[]= args[0].getBytes();
            // Update the data and digest it
            md.update(buf);
            digestStream.write(md.digest());

        } catch (Exception e) { 
            System.out.println(e);
        }
    }
}
```
2. Symmetric Ciphers

- A *symmetric cipher* in conjunction with a secret key translates *plaintext* to *ciphertext* (Secret-key cryptography)
- Cipher can also recover plaintext from ciphertext using the *same* key
- Both encryption and decryption use the same key
- Formally
  - $E_k(M) = C$, where $M$ is the *plaintext*, $C$ is the *ciphertext* and $k$ is the key
  - $D_k(C) = M$, where $C$, $M$ and $k$ have the same meaning
- The essential property: $D_k(E_k(M)) = M$
Symmetric Ciphers (cont’d)

- Disadvantages
  - Need secure method to transfer key
  - No authentication because same key used on both ends
  - Sender needs separate secret key for each receiver

- Key distribution center (KDC)
  - Shares secret key with users in network
  - Encrypts session key with secret keys to sender and receiver
  - Session key used for transaction
  - New keys and less couriers for transactions, but security depends on security on KDC
Symmetric Ciphers (cont’d)

Encrypting and decrypting a message using a symmetric secret key
Symmetric Ciphers and KDC

Distributing a session key with a key distribution center
Symmetric Ciphers (cont’d)

Types of symmetric ciphers:

- *Block ciphers* operate on a group of bits. The same plaintext block will encrypt to the same ciphertext block when using the same key.

- *Stream ciphers* operate on the stream of bits or bytes. They produce always different ciphertext.

Most block algorithms obey the Feistel Network property (algorithms for encryption/decryption are the same)
Implementations

- **Data Encryption Standard (DES)**
  - Uses *block cipher*: Creates bit groups from message and applies algorithm to whole block
  - DES standard set by *American National Standards Institute (ANSI)* for years, no longer considered secure

- Triple DES (3DES) replaced DES
  - Three DES systems in row with unique secret key

- **Advanced Encryption Standard (AES)** is new standard
  - *Nation Institute of Standards and Technology (NIST)* currently evaluating *Rijndael* for AES
3. Asymmetric Ciphers

- Uses *public-key* (distributed) and *private-key* (kept secret)
- Public-key decrypts private-key and vice-versa
- Computationally infeasible to deduce private-key from public-key
- Authentication
  - If receiver’s public-key and sender’s private key are both used, both parties are authenticated
- *RSA*: most common public-key algorithm
  - Used by most Fortune 1000 and e-commerce businesses
Asymmetric Ciphers

- Asymmetric ciphers involve the use of different keys for encryption/decryption:
  - $E_{k1}(M) = C$, where $k1$ is the encryption key
  - $D_{k2}(C) = M$, where $k2$ is the decryption key
- Essential property: $D_{k1}(E_{k2}(M)) = M$
- $k1$ and $k2$ are mathematically related and they are referred as the *public and private keys*
Asymmetric Ciphers

- Security is determined by the strength of the algorithm and the key’s length
  - Assume there is a computer capable of trying a billion keys per second
    - Key of 16 bits, $2^{16}$ possibilities, easy to break
    - Key of 128 bits, $10^{22}$ years to try all possibilities
- Use:
  - Public-key cryptography
    - E.g. SSL
  - Digital signatures
  - Certificates
  - Pretty Good Privacy (PGP), encrypts e-mails and files using “web of trust”
Public-key Cryptography (cont’d)

Encrypting and decrypting a message using public-key cryptography.
Authentication with a public-key algorithm
Key Management

Secrecy of private keys crucial to system security

- *Poor key management*: mishandling of private keys
- *Key generation*: process by which keys created
  - Should be as random as possible
- *Brute-force cracking*: decrypting message using every possible decryption key
Java Cryptography Extension (JCE)

- provides Java applications with various security facilities
- supports
  - secret-key encryption
    - 3DES
  - public-key algorithms
    - Diffie-Hellman
    - RSA
- customizable levels of encryption through
  - multiple encryption algorithms
  - various key sizes
- architecture is provider-based
  - developers add algorithms by adding providers’ algorithms
public class Encipher {

    private static final byte[] salt = {
        (byte)0xf5, (byte)0x33, (byte)0x01, (byte)0x2a,
        (byte)0xb2, (byte)0xcc, (byte)0xe4, (byte)0x7f
    };
    private int iterationCount = 100; // iteration count
    String password = "abc123";

    public Encipher() {
        Security.addProvider(new SunJCE());
        String line=null;
        StringBuffer buffer= new StringBuffer();
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            try { line = in.readLine(); } catch(Exception e){}
            if(line.equals("")) break;
            buffer.append(line + "\n");
        }
        String originalText = buffer.toString();
    }
}
Cipher cipher = null;
try {
    PBEKeySpec keySpec = new PBEKeySpec( password.toCharArray() );
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance( "PBEWithMD5AndDES" );
    SecretKey secretKey = keyFactory.generateSecret( keySpec );
    PBEParameterSpec parameterSpec = new PBEParameterSpec( salt, iterationCount );
    cipher = Cipher.getInstance( "PBEWithMD5AndDES" );
    cipher.init( Cipher.ENCRYPT_MODE, secretKey, parameterSpec );
} catch ( Exception e ) {}

byte[] outputArray = null;
try {
    outputArray = originalText.getBytes( "ISO-8859-1" );
} catch ( Exception e ) {}

CipherOutputStream out = new CipherOutputStream( System.out, cipher );
try {
    out.write( outputArray );
    out.flush();
    out.close();
} catch ( Exception e ) {}

public static void main( String[] args )
{
    Encipher crypto = new Encipher();
}
}
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.spec.*;
import com.sun.crypto.provider.SunJCE;
import javax.swing.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Decipher {

    private static final byte[] salt = {
        (byte)0xf5, (byte)0x33, (byte)0x01, (byte)0x2a,
        (byte)0xb2, (byte)0xcc, (byte)0xe4, (byte)0x7f
    };
    private int iterationCount = 100;   // iteration count
    String password = "abc123";

    public Decipher() {
        Security.addProvider( new SunJCE() );
        Vector fileBytes = new Vector();
        Cipher cipher = null;
        try {
            PBEKeySpec keySpec = new PBEKeySpec( password.toCharArray() );
            SecretKeyFactory keyFactory = SecretKeyFactory.getInstance( "PBEWithMD5AndDES" );
            SecretKey secretKey = keyFactory.generateSecret( keySpec );
            PBEParameterSpec parameterSpec = new PBEParameterSpec( salt, iterationCount );
            cipher = Cipher.getInstance( "PBEWithMD5AndDES" );
            cipher.init( Cipher.DECRYPT_MODE, secretKey,
                         parameterSpec );
        } catch ( Exception e ) {}
try {
    CipherInputStream in = new CipherInputStream(System.in, cipher);
    byte contents = (byte) in.read();
    while (contents != -1) {
        fileBytes.add(new Byte(contents));
        contents = (byte) in.read();
    }
    in.close();
}

catch (Exception e) {}
Run the example

- The secret key was predefined in Encipher.java and Decipher.java
- Create a plain text file “plaintext.txt” with the source data
- To encode:
  ```
  cat plaintext.txt | java Encipher > ciphertext.txt
  ```
  ciphertext.txt now contains the encoded text
- To decode:
  ```
  cat ciphertext.txt | java Encipher
  ```
Key Agreement Protocols

- Public-key algorithms not efficient for large amounts of data
  - Large computing power requirements slow communication

- Key Agreement Protocol
  - Two parties exchange keys over unsecure medium
  - *Digital envelope*: symmetric secret key encrypted using public-key encryption
Digital Envelope
Digital Signatures

- Provide proof of authenticity of the sender and integrity of the message
- The sender cannot deny that he/she signed a document (non-repudiation)
- Rely on public-key cryptography

The basic digital signature protocol is:

- The sender encrypts the document with his/her private key, implicitly signing the document
- The message is sent
- The receiver decrypts the document with the sender's public key, thereby verifying the signature
Digital Signatures (cont’d)

- To reduce processing time, often only a hash of the message is signed:
Digital Signatures (cont’d)

- Encryption can be included to guarantee confidentiality:
Public Key Infrastructure (PKI)

- Integrates public-key cryptography with *digital certificates* and *certification authorities* (CA’s)
  - Digital certificate: identifies user, issued by certification authority (such as VeriSign)
  - Digital certificates stored in *certificate repositories*

- *Certificate authority hierarchy*
  - *Root certification authority*, the Internet Policy Registration Authority (IPRA), signs certificates for *policy creation authorities* who set policies for obtaining digital certificates
  - Policy creation authorities sign for CA’s who sign for individuals and organizations
  - Signings use public-key cryptography
PKI, Certificates and CA (cont’d)

- Changing keys necessary for maintaining security
  - Digital certificates have expiration dates
  - Canceled and revoked certificates placed on certificate revocation list (CRL)

- Ensuring authenticity
  - Check certificate with CRL (inconvenient)
  - *Online Certificate Status Protocol (OCSP)* validates certificates in real-time

- PKI and digital certificate transactions are more secure than phone line, mail or even credit-card transactions
Certificates

- Issued by a CA
- Digitally signed by the CA
- Implicit assumption: CA’s signature is widely available and trusted
- Use X.509 format
X.509 Format

- Version and Serial Number
- Subject Name and affiliation
- Issuer Name
- Signature Algorithm
- Period of Validity
A Certificate Authority

A portion of the VeriSign digital certificate. (Courtesy of VeriSign, Inc.)
Java Security Architecture
Java Security

- Java code can originate and run anywhere in the network
- Java has been designed to run code securely via enforcement of security policies during execution
Evolution of Java Security

- JDK 1.0: The sandbox
  - The sandbox model confines Java applets to a defined arena where they cannot affect system resources
  - Applications enjoy unlimited access to all resources
Evolution of Java Security

- JDK 1.1: all or nothing
  - Introduced signed applets which enjoyed unlimited access to all resources just like local applications
  - No selective access
Evolution of Java Security

- JDK 2: fine-grained security
  - Flexible policy for applets and applications
  - Introduces the concept of ProtectionDomain
Java 2 Security Architecture

- Bootstrap class files
- System class files
- User class files
- CodeSource (URL, Certificates)
- Bootstrap ClassLoader
- System ClassLoader
- ClassLoader
- Bytecode Verifier
- SecurityManager
- Protection Domains
- AccessController
- Operating System
- Hardware
- Policy Database
- keystore
- Permissions
1. **Byte-code verifier**

- It screens the code to be sure that it was produced by a trustworthy compiler:
  - the format of the class file, the right length, the correct magic numbers, no operand stack overflows and underflows, and so on.
  - confirms or denies that the class file is consistent with the specifications
- Its behavior may be altered with command line options on the interpreter, when applicable.
2. ClassLoader

- The ClassLoader loads Java byte codes into the JVM
- Works in conjunction with the SecurityManager and the access controller to enforce security rules
- Information about the URL from which the code originated and the code's signers is initially available to the ClassLoader
3. CodeSource

- The object `java.security.CodeSource` fully describes a piece of code:
  - code's origin (URL)
  - digital certificates containing public keys corresponding to private keys used to sign the code.
- Many access-control decisions are based in part on this property.
4. Protection domains

- It is more flexible to group classes into protection domains and associate permissions with those domains (Rather than to associate permissions to individual classes)
- This relationship between the class and the permissions via the protection domain provides for flexible implementation mechanisms.
5. Policy

- The numerous mappings of permissions to classes are collectively referred to as *policy*.
- A policy file is used to configure the policy for a particular implementation.
- It can be composed by a simple text editor or using policytool (GUI).
6. Permissions

- Permission classes represent access to various system resources such as files, sockets, and so on.
- For example, permission may be given to read and write files in the /tmp directory.
- Permission classes are additive. They represent approvals, but not denials.
- A number of permission classes are subclasses of the abstract java.security.Permission class, examples of which include FilePermission, AWTPermission, and even customized protections like SendMailPermission.
7. SecurityManager

- The class java.lang.SecurityManager is at the focal point of authorization.
- SecurityManager consists of a number of check methods. For example:
  - checkRead (String file) can determine read access to a file.
  - checkPermission(Permission perm, Object context) method can check to see if the requested access has the given permission based on the policy.
- The access controller will raise an exception if the requested permission cannot be granted.
8. AccessController

- The java.security.AccessController class is used for three purposes:
  - To decide whether access to a critical system resource should be allowed or denied, based on the security policy currently in effect
  - To mark code as privileged, thus affecting subsequent access determinations
  - To obtain a snapshot of the current calling context, so access-control decisions from a different context can be made with respect to the saved context

- While the SecurityManager can be overridden, the static methods in AccessController are always available
9. keystore

- The keystore is a password-protected database that holds private keys and certificates.
- A password is selected at the time of creation.
- Each database entry can be guarded by its own password for extra security.
- Certificates accepted into the keystore are considered to be trusted. Keystore information can be used and updated by the security tools provided with the SDK.
import java.io.*;
import java.util.*;

public class writeFile {
    public writeFile() {
        String filename="thisisthefile.txt";
        File file = new File(filename);
        try {
            BufferedWriter output = new BufferedWriter(new FileWriter(file));
            output.write("Hello there");
            output.close();
        } catch (SecurityException e) {
            System.err.println("writeFile: caught security exception");
        } catch (IOException e) {
            System.err.println("writeFile: caught IO exception");
        }
    }

    public static void main(String[] args) {
        writeFile wf = new writeFile();
    }
}
Running the Example

- This succeeds:
  - java writeFile

- This produces a security exception:
  - java –Djava.security.manager writeFile
Defining the policy

- Create the file my.policy:
  
  ```
  grant {

    permission java.io.FilePermission "<<ALL FILES>>", "write";

  };
  ```

- Now run the program:
  
  ```
  java -Djava.security.manager -Djava.security.policy=my.policy writeFile
  ```
import java.io.*; import java.util.*; import java.awt.*; import java.applet.*;

public class writeFile extends Applet {
    public void paint(Graphics g) {
        String filename="thisisthefile.txt";
        File file = new File(filename);
        try {
            BufferedWriter output = new BufferedWriter(new FileWriter(file));
            output.write("Hello there");
            output.close();
            g.drawString("File " + filename + " written", 10, 10);
        } catch (SecurityException e) {
            g.drawString("writeFile: caught security exception", 10, 10);
        } catch (IOException e) {
            g.drawString("writeFile: caught IO exception", 10, 10);
        }
    }

    public static void main(String[] args) {
        Frame f = new Frame("writeFile");
        writeFile wf = new writeFile();
        wf.start();
        f.add("Center", wf); f.setSize(300,300); f.show();
    }
}
<html>
<title>Java Security Example: Writing Files</title>
<h1>Java Security Example: Writing Files</h1>
<hr>
<APPLET CODE=writeFile.class WIDTH=500 HEIGHT=50>
</APPLET>
<hr>
</html>
Running the Example

- This produces a security exception:
  appletviewer index.html

- This succeeds:
  appletviewer –J"-Djava.security.manager=my.policy" index.html
Browsers and Security

- Default lack of trust in downloaded code
  - Addressed by the sandbox model
- Limited access to command-line options within the browser
  - No simple way to deploy and use customized policy files
- Inadequate support for some security features in the JVMs bundled with browsers
  - Solved by using a java plug-in
SDK Security Tools

- **Keytool**
  - Manages keystores and certificates

- **Jarsigner**
  - Generates and verifies JAR signatures

- **Policytool**
  - Manages policy files via a GUI-based tool
keytool

- Create/Manage public/private key pairs
- Issue certificate requests (sent to the appropriate Certification Authority)
- Import certificate replies (obtained from the Certification Authority you contacted)
- Designate public keys belonging to other parties as trusted
keytool

- **Keystore**
  - repository for storing public and private keys
  - modifying stored keys requires use of password
  - default keystore located in `home/user/.keystore`

- **command line arguments**
  - `-genkey`
    produces private and public key pair
  - `-export`
    export a certificate
  - `-import`
    import certificate from trusted source
  - `-list`
    list all contents of keystore
  - `-alias <alias_name>`
    identify public and private pair for later use
keytool-generated certificates identified through

- commonName (CN)
- organizationUnit (OU)
- organizationName (O)
- localityName (L)
- stateName (S)
- country (C)
To generate a public and private key pair

    keytool -genkey -alias MyCertificate

Obtain digital certificate from certificate authority

    keytool -certreq -alias MyCertificate -file myRequest.cer

Submit certificate file to authority

    follow authority’s steps on Web site

To generate certificate other users may use

    keytool -export -alias MyCertificate -file myCertificate.cer
Digital Signatures for Java Code

- Java Plug-in supports RSA-signed applets

Steps

- generate RSA keypair
  \[\text{keytool} \ -\text{genkey} \ -\text{keyalg} \ RSA \ -\text{alias} \ MyCertificate\]

- export digital signature to file
  \[\text{keytool} \ -\text{export} \ -\text{alias} \ MyCertificate \ -\text{file} \ myCertificate.cer\]

- add to keystore
  \[\text{keytool} \ -\text{import} \ -\text{alias} \ MyTrustedCertificate \ -\text{keystore} \ cacerts \ -\text{file} \ myCertificate.cer\]
  - cacerts is complete path to keystore

- sign applet’s JAR file with digital signature
  \[\text{jarsigner} \ \text{FileTreeApplet.jar} \ MyCertificate\]

- enable Java Plug-in instead of Web browser’s JVM
  \[\text{htmlconverter} \ \text{signedApplet.html}\]
Example

- See LectureSet6/applet_signature
- Server side:
  - `keytool -genkey -alias alias -keystore server.ks -storepass storepass -keypass keypass`
  - `keytool -selfcert -alias alias -keystore server.ks -storepass storepass -keypass keypass`
  - `keytool -export -file client.cer -alias alias -keystore server.ks -storepass storepass -keypass keypass`
  - `keytool -list -keystore server.ks -storepass storepass -keypass keypass`
  - `jarsigner -keystore server.ks -storepass storepass -keypass keypass -rf WriteFile.jar rlent`
Example: Client side

- Using appletviewer:
  - `keytool -printcert -file client.cer`
  - `keytool -import -file client.cer -keystore client.ks -storepass storepass -keypass keypass`
  - `appletviewer -J-Djava.security.policy=client.policy index.html`

- Using a browser
  - Install Java plug-in!
Authentication

- Current authentication models
  - restrict access to certain aspects of a program
  - allow users to connect to a network
  - regulate resources available to users on network

- Java Authentication and Authorization Service (JAAS)
  - based on plug-in framework
  - allows Kerberos and single sign-on implementations
Kerberos

- Employs secret key cryptography
- Authentication handled by
  - Kerberos system
    - authenticates client’s identity
  - secondary *Ticket Granting Service* (TGS)
    - similar to key distribution centers
    - authenticates client’s rights to access services
- Authentication cycle
  1. client submits user name and password to Kerberos server
  2. server returns Ticket-Granting Ticket (TGT)
     - encrypted with client’s key
  3. client decrypts TGT
  4. client requests *service ticket* by sending decrypted TGT to TGS
  5. server authorizes client with renewable service ticket
Single Sign-On

- Single sign-on allows users to log into different servers once with single password.
- three types:
  1. workstation login scripts
     - login script sends password to each application
       - stores password on workstation
  2. authentication server scripts
     - authenticate users with central server
  3. tokens
     - once authenticated, non-reusable token identifies user
Java Authentication and Authorization Service (JAAS)

- Protects applications from unauthorized users.
- Based on *Pluggable Authentication Module (PAM)*
  - supports multiple authentication systems
  - different authentication systems may be combined
- Can control access by
  - user
    - governs access to resources on user policies
  - group
    - associates user to group, bases policies on group privileges
  - role-based security policies
    - similar to group policies
    - unlike group policies, no default policies exist
      - users obtain privileges to needed applications based on intended task
Example **AuthenticateNT**

To execute:

```
java -Djava.security.policy=java.policy
    -Djava.security.auth.policy=jaas.policy
    -Djava.security.auth.login.config=jaas.config
AuthenticateNT
```
Secure Sockets Layer (SSL)

- Nonproprietary protocol
- Used to secure communications between computers
- Implements
  - public-key technology using RSA algorithm
  - digital certificates
    - to authenticate server
    - to protect private information
- Does not require user authentication
TCP/IP and SSL Protocol Stack

1. Application Layer (HTTP)
2. SSL
3. TCP
4. IP
SSL (cont’d)

Process:
1. client sends message to server
2. server responds with digital certificate
3. client and server negotiate session keys
   • use public key cryptography for negotiation
4. once keys established, communication proceeds
   • information encrypted
   • information transmitted
   • information decrypted at receiving end

Primarily secure *point-to-point* connections
Java Secure Socket Extension (JSSE)

- SSL encryption integrated into Java through *Java Secure Socket Extension (JSEE)*
- Secures passage of information between two clients
- Use of SSL connections transparent to user
SSL Handshake Protocol

1: ClientHello
2: ServerHello
3: Certificate (optional)
4: Certificate Request (optional)
5: Server Key Exchange (optional)
6: ServerHelloDone
7: Certificate (optional)
8: Client Key Exchange
9: Certificate Verify (optional)
10: Change Cipher Spec
11: Finished
12: Change Cipher Spec
13: Finished
14: Encrypted Data
Example SSL Client/Server

~/LectureSet6/ssl, files:
- client/client.java
- server/server.java
Run the example: Server side

- Create keystore and certificate:
  ```
  keytool -genkey -keystore SSLStore -alias SSLCertificate -keystore pass keystore pass -storepass storepass
  ```
  If SSLStore does not exist, this will create keystore with storepass as password

- To check stored entries:
  ```
  keytool -list -keystore SSLStore
  ```

- Execute `sslServer`
  ```
  java -Djavax.net.ssl.keyStore=SSLStore
    -Djavax.net.ssl.keyStorePassword=password sslServer
  ```

- Export Certificate
  ```
  keytool -export -alias SSLCertificate -keystore SSLStore -file mycertificate.cer
  ```

Now make file mycertificate.cer available to client
Run the example: Client side

- Get file mycertificate.cer from server
- Import Certificate
  
  ```
  keytool -import -alias SSLCertificate -keystore SSLStore -file mycertificate.cer
  ```

- To check the entry:
  
  ```
  keytool -list -keystore SSLStore
  ```

- Execute sslClient
  
  ```
  java -Djavax.net.ssl.trustStore=SSLStore
  -Djavax.net.ssl.trustStorePassword=password sslClient
  ```
Example HTTPS

~/LectureSet6/https, files:
- sslWebClient.java

- Two versions of the server:
  - sslWebServer.java (security parameters externally defined)
  - sslWebServer2.java (security parameters internally defined)

- cert-s.sh and cert-c.sh create certificates for server and client
- runs.sh and runc.sh execute server and client with external parameters